Experience with model-based performance, reliability and adaptability assessment of a complex industrial architecture

Dominguez Gouvêa, Daniel ; de A Assis D Muniz, Cyro ; Pinto, Gilson A ; Avritzer, Alberto ; Leao, Rosa Maria Meri ; de Souza e Silva, Edmundo ; Diniz, Morganna Carmem ; Berardinelli, Luca ; Leite, Julius C B ; Mossé, Daniel ; Cai, Yuanfang ; Dalton, Mike ; Happe, Lucia ; Koziolek, Anne

Abstract: In this paper, we report on our experience with the application of validated models to assess performance, reliability, and adaptability of a complex mission critical system that is being developed to dynamically monitor and control the position of an oil-drilling platform. We present real-time modeling results that show that all tasks are schedulable. We performed stochastic analysis of the distribution of task execution time as a function of the number of system interfaces. We report on the variability of task execution times for the expected system configurations. In addition, we have executed a system library for an important task inside the performance model simulator. We report on the measured algorithm convergence as a function of the number of vessel thrusters. We have also studied the system architecture adaptability by comparing the documented system architecture and the implemented source code. We report on the adaptability findings and the recommendations we were able to provide to the system's architect. Finally, we have developed models of hardware and software reliability. We report on hardware and software reliability results based on the evaluation of the system architecture.

DOI: https://doi.org/10.1007/s10270-012-0264-x

Noname manuscript No.
(will be inserted by the editor)

Experience with Model-based Performance, Reliability and Adaptability Assessment of a Complex Industrial Architecture

Daniel Dominguez Gouvêa, Cyro de A. Assis D. Muniz, Gilson A. Pinto
Alberto Avritzer
Rosa Maria Meri Leão, Edmundo de Souza e Silva
Morganna Carmem Diniz
Vittorio Cortellessa, Luca Berardinelli
Julius C. B. Leite
Daniel Mossé
Yuanfang Cai, Mike Dalton
Lucia Happe, Anne Koziolek

the date of receipt and acceptance should be inserted later

Abstract In this paper, we report on our experience with the application of validated models to assess performance, reliability, and adaptability of a complex mission critical system that is being developed to dynamically monitor and control the position of an oil-drilling platform. We present real-time modeling results that show that all tasks are schedulable. We performed stochastic analysis of the distribution of tasks execution time as a function of the number of system interfaces. We report on the variability of task execution times for the expected system configurations. In addition, we have executed a system library for an important task inside the performance model simulator. We report on the measured algorithm convergence as a function of the number of vessel thrusters. We have also studied the system architecture adaptability by comparing the documented system architecture and the implemented source code. We report on the adaptability findings and the recommendations we were able to provide to the system's architect. Finally, we have developed models of hardware and software reliability. We report on hardware and software reliability results based on the evaluation of the system architecture.

Daniel Dominguez Gouvêa, Cyro de A. Assis D. Muniz, Gilson A. Pinto
Chemtech a Siemens Company, RJ 20011-030, Brazil

Alberto Avritzer
Siemens Corporate Research, Princeton, NJ 08540

Rosa Maria Meri Leão, Edmundo de Souza e Silva
Federal University of Rio de Janeiro, COPPE, RJ 21941-972, Brazil

Morganna Carmem Diniz
Federal University of the State of Rio de Janeiro, RJ, Brazil

Vittorio Cortellessa, Luca Berardinelli
University of L' Aquila, Italy

Julius C. B. Leite
Universidade Federal Fluminense, Niterói, Brazil

Daniel Mossé
University of Pittsburgh, Pittsburgh, PA

Yuanfang Cai, Mike Dalton
Drexel University, Philadelphia, PA

Lucia Happe
Karlsruhe Institute of Technology, Karlsruhe, Germany

Anne Koziolek
University of Zurich, Zurich, Switzerland

1 Introduction

In this paper, we present our experience with the application of performance, reliability, and architecture modeling approaches to the assessment of a Dynamic Positioning System (DPS) architecture. The DPS under study is a software prototype that has been in development at Siemens-Chemtech for several years and is targeted to be deployed to control large mission-critical vessels. Its main task is to use the vessel's thrusters to control the vessel position and heading. Specifically, it is being designed to be deployed for monitoring and controlling deep-water oil-drilling vessels. These systems have very stringent performance and reliability requirements that need to be demonstrated prior to obtaining the required quality certifications. Therefore, mod-

eling of performance and reliability is a very important project objective.

The main contribution of this paper is the presentation of a detailed experience report of the application of several complementary performance, reliability and adaptability models to the architecture assessment of a complex mission-critical system. This experience report illustrates the breadth and depth of expertise required to model performance, reliability and architecture of large industrial systems.

In a companion paper [14], we have presented a new architecture review process that used a globally distributed review team to perform architecture risk assessment of this system. We employed a team of experts to identify and categorize the architecture risks related to the performance, reliability and adaptability non-functional requirements. The results presented in [14] were based on teleconference discussions and face-to-face interviews of the architects and domain experts.

In contrast, in this paper we present experimental results that were obtained by modeling performance, reliability and adaptability using data derived from the implementation of the DPS system architecture. Specifically, the performance modeling results presented in this paper were based on actual measurements performed on the implemented software prototype. The measurement results were analyzed and used to calibrate the models. In one instance, the actual implemented software library was executed inside the simulation model. The reliability modeling results presented for the hardware and the software reliability assessments were based on the analysis of the system architecture using hardware failure rates obtained from the hardware vendors and from the black-box software execution in a simulated environment. The adaptability modeling approach was performed by generating Design Structure Matrixes (DSMs) from the Enterprise Architect (EA) documentation tool and from the actual DPS prototype source code. This paper extends a previous publication [20] by providing a black-box reliability analysis and extending the adaptability analysis taking into account design structure. Furthermore, the lessons learned were extended with recent insights.

In summary, we have built several models to assess different aspects of the DPS architecture. These models were instrumented by measuring the performance of parts of the implemented software. The execution of the models provided both positive and negative feedback to the project. As a result of this effort, the project learned that the tasks as implemented could be proven to be schedulable. In addition, the sensitivity analysis showed that the system would perform well even for vessels with a larger number of thrusters. The evaluation of convergence characteristics of the thruster allocation under six scenarios that were designed by domain experts was beneficial as it showed good convergence for these scenarios. The difference in convergence behavior for different number of thrusters indicates a need to test additional scenarios before a production version of the DPS system is certified for production deployment. The framework developed for the thruster allocation evaluation is very efficient and could be used to test hundreds of scenarios. The adaptability assessment evaluation uncovered several discrepancies between documentation and implementation. We were able to provide good feedback to the project about the need to continuously maintain the architecture documentation up to date.

The outline of the paper is as follows. In Section 2 we present an overview of the work related to the modeling approaches used in this paper. In Section 3 we present an overview of the Dynamic Positioning System architecture and the information flow from the sensors to the thrusters. We describe the most important task types, their responsibilities, and how these tasks are activated. In Section 4 we present the three different approaches that were used to analyze system performance: worst case analysis *Real-Time* modeling, *Stochastic Modeling*, and *Tangram-II* actual implementation simulation modeling. The objectives of the three different performance modeling approaches are:

1. In the *Real-Time* modeling performance subsection, we report on the results that were obtained from the DPS prototype system. Data was collected on 1,000,000 execution instances, for three different vessel configurations. The worst-case execution times were computed and we were able to show that the system is schedulable under the Rate Monotonic Scheduling discipline (RM),
2. In the *Stochastic Modeling* performance sub-section, we report on experiments that were conducted using a *Palladio Component Model (PCM)* based high-level simulation model. This model was instrumented using data that was obtained from measurements of the system tasks execution time. The objective of running experiments using the *Palladio Component Model (PCM)* was to understand the full-distribution of tasks execution time and to assess the impact of number of sensors and thrusters on the tasks execution time.
3. In the *Tangram-II* implementation based modeling approach the actual library implemented for the thruster allocation was executed inside the Tangram-II model to evaluate some of the thruster allocation important characteristics such as the number of iterations required for convergence, and

the force distribution among the different thrusters as a function of the number of thrusters actually used by the vessel. The framework created for running the library is very useful as it allows for a controlled and efficient execution of the thruster allocation model.

In Section 5 we present results of our experiments using an architecture adaptability modeling approach. We compared source code based and design based Design Structure Matrixes (DSNs), and we report on our experience with adaptability assessment of the system architecture. In Section 6 we present software reliability experimental results based on the black-box execution of the complex industrial system and an additional Tangram-II model representing hardware reliability. The hardware reliability model was created using the system architecture and it was instrumented with hardware failure data. In the software reliability part of Section 6 we present cumulative failure data obtained from the execution of the complex industrial system. Section 7 contains our conclusions and lessons learned.

2 Related Work and Background

In this section we present a brief overview of previous work that is related to the non-functional requirement modeling approaches used in this paper.

2.1 Architecture Assessment

In this section we present a brief overview of the Non-functional requirement modeling approaches used in this paper.

In this section we also summarize the architecture assessment process used for identifying the non-functional requirement models that were used to evaluate this complex industrial system. A detailed description of the process used and the architecture assessment results can be found in [14].

We used a globally distributed review team with performance analysis and architecture expertise in the different non-functional domains being probed, as for example, real-time, hardware and software reliability, performance, and architecture design structure.

The project quality attributes most important to the business were defined by the customer, and were later refined into operational scenarios by the domain experts. Several meetings were held to categorize the operational scenarios according to risk and the business utility, using an operational scenario-based analysis of the software architecture as described in [22].

The non-functional requirements analysis of the complex industrial system, as reported in [14], was focused on the following domains:

- *Adaptability* - the property of being able to add new features quickly to the system,
- *Reliability* - the probability of correct operation without failures for a given time under a specific environment,
- *Real-Time* - a property that enables the system architect to reason about the probability of tasks meeting hard deadlines,
- *Fault-Tolerance* - mechanisms to prevent faults from propagating to the service boundary,
- *Performance* - the ability to control the total response time of the tasks executing in a control loop.

The result of the scenario-based analysis process was the identification of the most important scenarios and the associated domains. The most important domains identified for further analysis were adaptability, robustness (reliability, fault-tolerance), performance, and real-time. The remainder of this paper presents the details of the non-functional requirement models that were developed as result of the architecture assessment of the complex industrial architecture.

2.2 Real-Time

The periodic real-time scheduling problem has been studied extensively for uniprocessors. There are two types of task priorities that can be considered in real-time systems, namely *dynamic priority* and *static priority*. In the former, two optimal dynamic scheduling algorithms have been devised on uniprocessors, namely *Earliest Deadline First* (EDF) and *Least Laxity First* (LLF). Both guarantee that each task meets its deadline if the sum of task utilizations is less than 100%, that is, if the processor is not overutilized. Formally, a set of n tasks will be accepted into the system, and guaranteed to meet all deadlines, if [25]:

$$\sum_{i=1}^{n} \frac{WCET_i}{T_i} \leq 1 \tag{1}$$

where WCET is the worst-case execution time of the task, T_i is the task period and n the number of tasks, then the system is guaranteed to be schedulable. The deadline of the tasks is considered to be equal to their periods. Note that $\frac{WCET_i}{T_i}$ represents the utilization of a task, or the percentage of the processor being used by task τ_i. Researchers have noted that some task sets scheduled by LLF may suffer from a high number of preemptions. Although EDF is an optimal scheduling

algorithm, it also has some overhead, due to the ordering of tasks by deadlines every period.

To avoid the overhead of dynamic priorities, static-priority algorithm Rate Monotonic Scheduling (RMS) is optimal among all fixed-priority scheduling algorithms. A fixed-priority algorithm such as RMS associates each task τ_i with a fixed priority $p_i \propto 1/T_i$, that is, the inverse of the period of the task, and executes the available task with the highest priority. The benefit of this scheme is that it is simple: tasks never have to change priority and they execute periodically at their own priority[1]; in addition, many operating systems (e.g., QNX) implement fixed priority schemes. The original admission control for RMS was proposed in [25] (and is very well described in [8]) and it accepts a set of n tasks into the system, and guarantees that all deadlines will be met if

$$\sum_{i=1}^{n} \frac{WCET_i}{T_i} \leq n(2^{1/n} - 1) \quad (2)$$

the system is guaranteed to be schedulable. The restriction expressed in Equation 2 is a sufficient condition. Note that Equation 2 has a smaller bound than Equation 1 due to the interference of the fixed priority tasks on one another: even though a task τ_i may have an earlier deadline than another task τ_j at a specific time instant, the priority of τ_j might be higher than that of τ_i, causing τ_i to have to wait for τ_j, requiring a smaller bound to account for such interference.

There are some restrictions on the task sets considered in these bounds (for all above algorithms and bounds), namely that it be preemptable (which most tasks are) and that each task is independent from each other. The use of shared resources among RT tasks may lead to *priority inversion* and subsequently to unbounded blocking times. Several resource access protocols such as the Priority Inheritance Protocol [29], the Priority Ceiling Protocol [29] and the Stack Resource Policy [3] have been proposed to solve this problem. The schedulability analysis for these protocols take the effect of shared resources into account by adding an extra term in the formula for a task response time that is at most $max_{i=1}^{n} WCET_i$ for higher priority tasks.

In our system, tasks are periodic (actually all tasks have the same period, making it even more efficient) but they do use shared resources, therefore making some tasks dependent on each other. A work around this restriction is due to the nature of the DPS system, with its large time delays for the physical movement of the vessels, which allows for stale data (1-2 periods old) to be used without changing significantly the physical characteristics of the system. In addition, in our system, the WCET of the tasks is very small in comparison to the period of the tasks, and adding one extra WCET would not make a big difference in the total utilization of the system, and would still keep it under the bound required for all tasks to meet their deadlines. Thus, we need not be concerned with the priority inversion problem.

Other utilization bounds have been proposed, for example in [7] the bound takes into account the tasks periods and exact admission controls are given in [2, 24]. Due to the nature and utilization of our system, these more advanced bounds need not be used.

[1] Tasks can also be dispatched by hardware, given the simple fixed-priority scheme.

2.3 Adaptability Model

As an emerging design model, *design structure matrix* (DSM) [4,30,10] has been used to visualize the dependency and modular structure within software systems. In this study, we use DSMs to model both designed architecture structure and its implementation for the purpose of assessing the system's adaptability when the software changes.

A design structure matrix is a square matrix where the columns and rows are labeled with the same set, in the same order, of design dimensions where decisions are needed. We refer to each design dimension as a *variable*. A variable can model a component in a design model, a class in source code, or a method within a class. If a cell in a DSM is marked, it means that the variable on the row depends on the variable on the column. A *module* in a DSM represents a group of variables, modeled as a block along the diagonal. For example, all the classes related to the same feature can be aggregated into one module.

To address the problem that manually constructing a DSM can be time-consuming and error-prone, Cai and Sullivan developed the *Augmented Constraint Network* (ACN) model to represent the dependency relations among design dimensions (variables) using logical expressions [9,10,33]. From an ACN, the semantics of *pair-wise dependency* can be formally defined and a DSM can be automatically derived. In order for designers trained with UML modeling to leverage these techniques, Cai and her students have formalized and implemented the transformation of prevailing design models and software artifacts, such as UML class diagram, UML component diagram and source code into ACN models [33]. After that, the structures of these artifacts can be automatically represented as DSMs.

To further analyze the modularity and adaptability of the system, we also clustered these DSMs into a special hierarchy, the *design rule hierarchy* (DRH) [32].

The top level of the hierarchy models most influential design decisions that only influence other parts of the system but are not influenced by them. For example, most systems applying architectural or design patterns, such as model-view-controller pattern, publisher-subscriber pattern, or abstract factory pattern, usually employ a group of key interfaces (e.g. an abstract factory interface) to lead the pattern. These interfaces are modeled as *design rules*–the design decisions that decouple the rest of the system into modules [4]. These design rules should remain stable because many other components depend on it.

In a DSM, these design rules are modeled as special variables that do not depend on non-design-rule variables, but only dominate (influence) other variables. In a DRH-clustered DSM, the subsequent layers contain design dimensions that only depend on the previous layers. The final layer contains modules that do not influence any other components of the system, and can be changed freely as long as the design rules before it are stable. Another feature of the hierarchy is that each layer contains a set of modules that are independent from each other.

DSM modeling makes it easy to assess the adaptability of the system. If an existing component needs to be changed, we just need to calculate which and how many other elements depend on it and hence may be subject to change. The fewer number of elements are affected, the more adaptable it is. If a new component is to be added or deleted, we can estimate at which level these components will be added or deleted. If these components only impact the bottom layer as independent modules, it means that the open to extension close to modification principle can be followed, and the system is easily adapted to the given change. On the other hand, if a change will affect the top level design rules, or will affect several other components, then the system is not very adaptive to the given changes. By comparing the designed and implemented modular structure, we can verify that the implementation realizes the designed adaptability.

Based on the design rule hierarchy, the overall adaptability can be assessed using the *independence level*[31] metric. This metric measures the percentage of the system contained in the bottom layer of a design rule hierarchy, the larger the portion, the more adaptable the system is because the more modules can be changed, or even substituted with new versions without influencing the rest of the system. By contrast, if the portion of the system in the bottom layer is small, it means that most of the system depend and influence each other, and changing part of it will often influences the many other elements.

2.4 Black-Box Software Reliability

In black-box software reliability estimation the system is executed as a whole, and the system failure rate is statistically estimated based on the measured failure data [18]. The execution of accurate black-box software reliability testing requires that software be completely developed and be stable. Black-box software reliability models are difficult to apply in practice because a production-like environment needs to be built with enough detail to reveal service impacting failures. In addition, the test environment needs to drive the system under test using a production-like operational profile, as described in [26]. Section 6.2 presents the black-box reliability estimation of the complex industrial system analyzed in this paper.

In the following sub-section we describe, for the sake of completeness, a software reliability estimation approach based on the analysis of software architecture. For a systematic review of architecture-based software reliability estimation approaches, see [19].

2.5 Component-Based Software Reliability Model

In contrast to the black-box software reliability estimation approaches, reliability models based on the software architecture need to be aware of the structure of the software components and the way they are interconnected. For this reason, even though such models are able to capture more details of a system, they are not always applicable, because several model parameters are either unknown, when the approach is applied early in the software lifecycle, or difficult to collect from running systems.

We describe in this section the preliminary models and the associated parameters required by the reliability analysis methodology described in [11].

In Figure 1 we give a visual representation of the parameters of our component-based software reliability model, as associated to model elements, that are:

- the *internal failure probability* (*intf(i)*) is the probability that a component generates a failure caused by some internal fault;
- the *error propagation probability* (*ep(i)*) is the probability that a component propagates to its output interface an erroneous input it has received;
- the *propagation path probability* (*p(i,j)*) is the transition probability from the output interface of component i to the input of component j.

The *REL* index represents the reliability of the system model in Figure 1, that is a function of the above parameters.

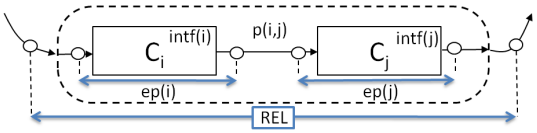


Fig. 1 Reliability attributes of a component-based model.

Values for these parameters can be estimated as follows:

- the *internal failure probabilities* can be roughly estimated from the KLOC (thousands of lines of code) of the corresponding source code artifacts. This and other estimation techniques are described in [15];
- the *error propagation probabilities* can be set: (i) to 0 for the components that lack error masking mechanisms, i.e. components that always propagate to their output interfaces erroneous inputs they receive, (ii) to 1 when the most effective error masking mechanism is implemented for a component so that no errors propagate to its output interface; (iii) all intermediate values (between 0 and 1) represent different capabilities of component error masking, and they can be estimated with different techniques (e.g. [1]);
- the *propagation path probabilities* can be derived at early design stages from the system models that are available at that time, or from software artifacts (e.g. UML interaction diagrams), possibly annotated with stochastic data applied on interaction patterns [12].

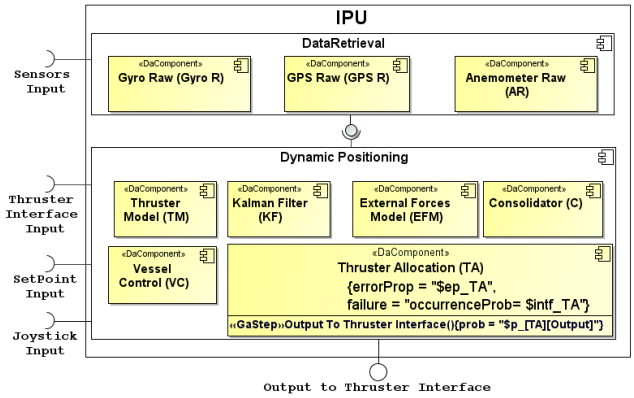


Fig. 2 The software architecture of DPS controller.

The UML Component Diagram in Figure 2 shows the software architecture of the DPS controller (aka IPU). These components are arranged according to the tasks they are involved in, namely *DataRetrieval* and *DynamicPositioning* (see Section 3).

The required reliability parameters (i.e., *intf(i), ep(i), p(i,j)*) are annotated as variables (i.e. $-prefixed terms in the figure) within *tags* of `stereotypes` (i.e., `DaComponent`::*failure*, `DaComponent`::*errorProb* and `GaStep`::*prob*, respectively). These stereotypes are defined in UML profiles suitably devised to support model-based analysis methodologies[2]. Details on how to compute the *REL* system reliability for such an architectural description, as a function of the above parameters, are provided in [11].

We could not apply the component-based reliability approach to the DPS system, because the DPS system testing infrastructure was not instrumented to collected the data required for the estimation required parameters. We retain relevant to briefly report on this experience, because it allows us to focus on the obstacles that can be encountered in this direction.

The DPS system simulation model kept track of messages and data exchanged among software components. However, no probes were implemented to collect data related to internal failures, error and path propagation probabilities, which are necessary to compute the component-based reliability. In addition, the DPS system simulation did not collect the error propagation probabilities due to the lack of a complete system specification oracle.

Our experience on DPS resulted in the definition of a hierarchy among parameters, according to the increasing complexity of the collection approaches. Specifically:

- *internal failure probabilities intf(i)*: data to collect for such parameters are only related to the component C_i, in that such probability can be locally computed as the number of deviations of component output from a given specification oracle;
- *propagation path probabilities p(i,j)*: data to collect for such parameters are related to the amount of messages exchanged between components C_i and C_j, hence they can be probed on the connector among each pair of interacting components;
- *error propagation probabilities ep(i)*: differently from the above parameters, the values of this one depend on the system execution as a whole (i.e. they cannot be detected locally). In fact *ep(i)* requires that, during the system execution/simulation, an oracle exists for each component that determines if the current input, though within the feasible range for the considered component, represents an error for the whole computation. This occurs if the current input is not the one that should be produced by the processing that precedes this component along

[2] MARTE (Modeling and Analysis of Real-Time and Embedded systems [27]) and its extension DAM (Dependability Analysis Modeling) [6]

the current path. Therefore, a whole system specification oracle is needed to detect the amount of erroneous inputs a certain component propagates ahead.

2.6 Palladio Component Model

The implementation of the Stochastic Performance Analysis approach is based on an architectural modeling language called *Palladio Component Model (PCM)* [5,21]. The PCM is a modeling language specifically designed for performance prediction of component-based systems, with an automatic transformation into a discrete-event simulation of generalized queuing networks. Its available tool support (PCM Bench) allows performance engineers to predict various performance metrics, including response time, throughput and resource utilization. The PCM has been designed for performance prediction of component-based business information systems, focusing on the *reusability* of performance specification in different contexts and on enabling stochastic performance analysis by providing *distribution functions* for performance metrics.

The PCM supports explicit modeling of different context influences on performance properties (such as operational profile, available external services, and used hardware platform) to achieve *reusable* specifications. The PCM component performance specifications can be parametrized for these context influences, so that they can be reused together with the component implementation in different contexts. For example, the resource demand of a component can be modeled as a function of the amount of processed data, so that the same component specification can be reused in different systems and settings with varying amount of data being processed. Similarly, the control flow within components is modeled parametrically and can depend upon input parameters or results of other called components.

The goal of PCM predictions is to provide *distribution functions* of performance metrics, which can be used to for example determine whether service level agreements such as "the systems responds within 5 second in 90% of all cases" under a given operational profile is fulfilled. Such percentile requirements cannot be answered with mean-value based prediction approaches, and are yet relevant for performance requirements because they better reflect the user perception of the system's performance.

In a PCM model, time consumptions of single tasks can be modeled as generalized distribution functions, approximated using stepwise functions as shown in Figure 3. Every type of distribution function can be approximated in this way. In the figure, the distribution

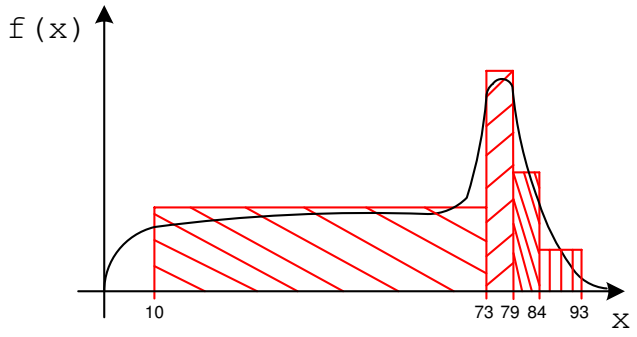


Fig. 3 Approximating a generalised distribution function with a stepwise defined function (from [23])

function, shown as a black probability density function, is approximated by the stepwise function, shown as red rectangles. The approximation is defined by defining a interval $]a, b]$ (e.g. from 10 seconds to 73 seconds) and the probability p that the random variable takes a value v with $a < v \leq b$ (e.g. 0.6 for the mentioned interval). Graphically, the area of each rectangle reflects the probability of the interval. The approximated distribution function is defined by enumerating the upper right corners of the red rectangles. Thus, accurate distributions of the overall performance metrics can be derived by simulation.

Such distribution functions are used to model the resource demands and other performance-relevant parameters of software components (cf. [5]). Several analysis approaches are supported to derive performance metrics of the overall system based on the such component performance models. In this work, we use the SimuCom simulation to retrieve performance metrics.

In addition to the PCM's primary domain of business information system, the main concept of reusable component performance specifications and prediction of distribution functions is relevant for other types of systems as well. In this work, we studied the applicability of the PCM in a system with real-time constraints. Our research questions was whether the abstractions used in PCM component performance models and analyses allow to answer relevant performance questions.

If combined with worst case analysis, a PCM prediction can give further insight into the distribution of response times. Potentially, the created component performance models can be reused in different contexts, e.g. in a product line of DPS systems for different ship configurations with varying hardware equipment and varying operational profile.

3 Dynamic Positioning System

The non-functional requirement models presented in this paper were applied to the Dynamic Positioning System (DPS) project. An overview of the DPS project architecture was presented in [14]. Figure 4 shows the physical architecture of the system. The system consists of sensors, controllers (aka IPU), human-machine interfaces (HMI) and thrusters. This paper focus is in the system's main flow, where the IPU plays the major role.

The IPU houses the system core, being responsible for consolidating the data from all sensors, computing the force needed to keep the vessel in the desired position, commanding the thrusters, and making important process data available. The IPU uses the QNX Neutrino Real-Time Operating System [28]. There are four kinds of tasks running in the IPU:

1. The *DataRetrieval* task (DR) is responsible for getting data from a specific sensor. There are three kinds of sensors in the system, each one of them with triple redundancy, which gives a total of 9 *DataRetrieval* tasks,
2. The *DataLayer* task (DL) stores the real-time values collected from sensors, from control tasks, and from the operator. The *DataLayer* task makes the collected data available for every other task that needs to access this data. It is essentially a memory-resident database that can be used to synchronize the tasks (passing information from one task to another),
3. The *DynamicPositioning* task (DP) is the main system task. It is responsible for running the mathematical algorithms [17] that compose the dynamic positioning system (in particular the thruster allocation algorithm), and also for sending commands to the thruster system,
4. The *IPUManager* task (IPUM) is responsible for coordinating the IPU redundancy. The *IPUManager* is responsible for selecting one and only one IPU to be the master, leaving the other two as backup stations. Since this selection is done periodically it is also used to keep all three IPUs synchronized. However, we do not consider the IPUM task in the timing analysis since it has a very low execution time.

The IPU information flow is illustrated in Figure 5. A *DataRetrieval* task is instantiated for each sensor that is connected to the system. The measurements collected by the *DataRetrieval* tasks are transferred to the *DataLayer* task and are routed to the *DynamicPositioning* task.

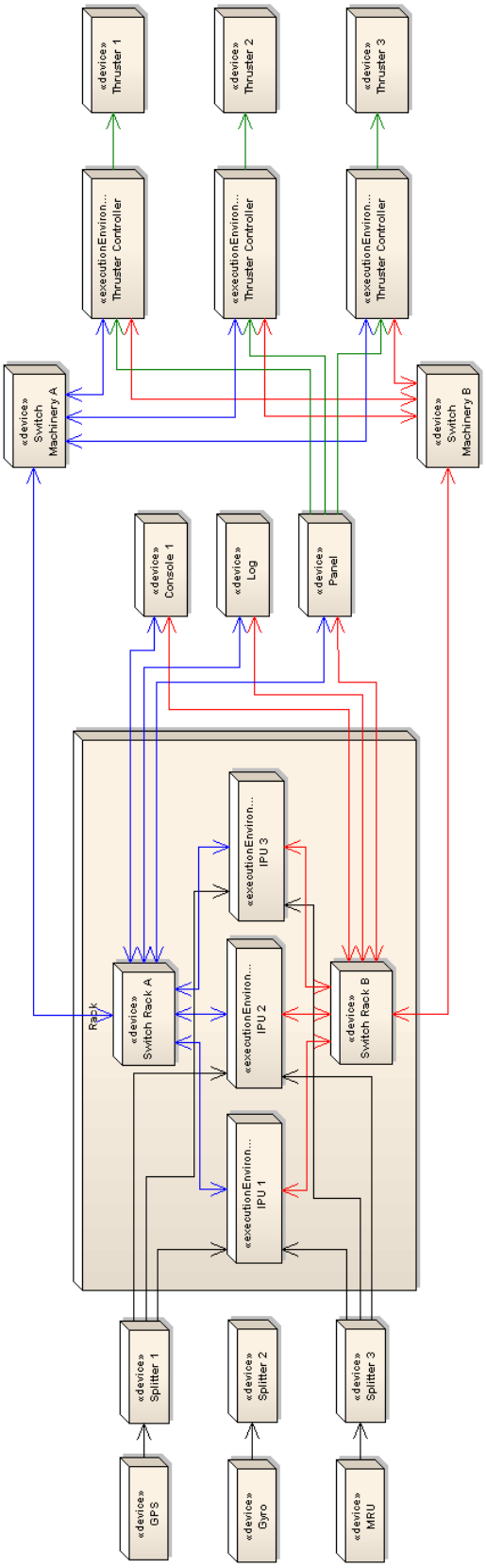


Fig. 4 Architecture overview of the Dynamic Positioning System

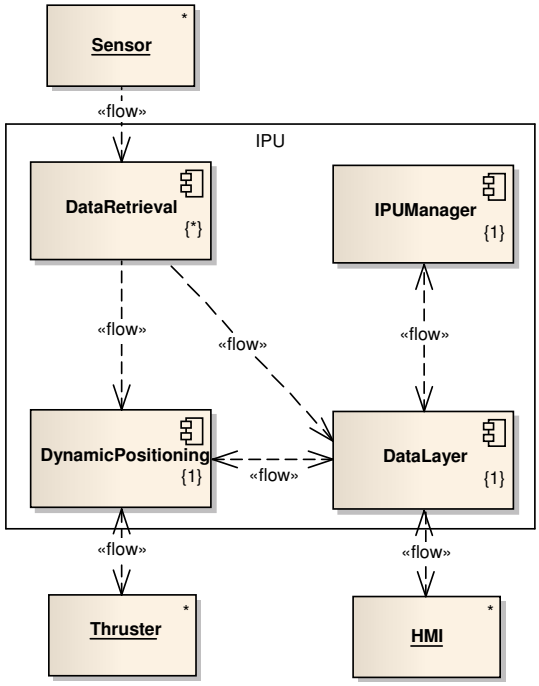


Fig. 5 IPU information flow

The control loop is the flow that begins in the Sensors, passes through the *DataRetrieval* and the *DynamicPositioning* tasks and ends in the Thrusters. The diagram in Figure 6 shows the relationships between the tasks.

4 Performance

In this section we present the three performance modeling approaches that were developed to assess real-time performance, task execution time as a function of the number of system interfaces, and the thruster allocation convergence characteristics.

The real-time task model level of abstraction assesses the impact of worst case control loop execution time on the system ability to satisfy the real-time requirement, as the the control loop must be executed in one second. The stochastic analysis model studies the impact of the number of sensors and thrusters on the control loop execution time distribution. The actual implementation approach executes the thruster allocation module inside a performance model to assess the impact of the number of thrusters on the convergence characteristics of the thruster allocation library, because the most critical task in the control loop is the thruster allocation algorithm. The thruster allocation algorithm is based on an iterative solution of an optimization problem. It is a small part of the control loop, but the investigation of the convergence characteristics of the thruster allocation algorithm is a very important modeling objective as bad convergence characteristics could have a significant impact on the ability of the DP system to control the vessel's position.

4.1 Real-time Analysis

Real-time is a property that allows reasoning about time and temporal characteristics of the system. In particular, for this DP system, the approach adopted in the DP architecture to implement real-time was to use a periodic task set that repeats execution at specific moments in time. The fixed-priority scheduling discipline used is known as Rate Monotonic Scheduling [8].

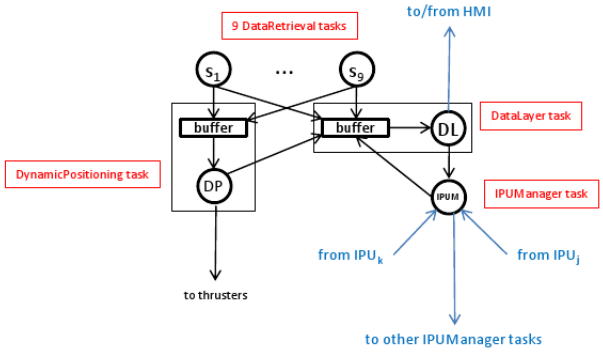


Fig. 6 DP tasks relationships

All the described tasks are periodic, with a $1s$ period, and this period was chosen also as the cycle execution. Experiments were conducted with a prototype system that executes each of the tasks individually and independently, and measures the execution time (ET) of a task for each instance executed. We measured the ETs of each task from 1,000,000 executions instances, for 3 different vessel configurations. These configurations are based on the number of thrusters in the vessel, and we used typical values of 4, 6, and 8 thrusters. In all cases, only the DP task has different worst-case execution time (WCET), DL and DR keeping the same execution times. The DP WCETs obtained were, approximately, $106.0ms$, $107.8ms$, and $168.9ms$, for 4, 6, and 8 thrusters, respectively. In Table 1, we show the worst-case, the average case and other statistical measures of the ETs of the tasks, for an 8 thrusters configuration, i.e., the most demanding one.

As indicated in Table 1, in 99% of the cases the execution times are well below the WCETs, and this indicates the existence of an additional spare capacity for running background (non-critical) tasks. We note that even if there is a major reduction in execution times of

Table 1 Execution times (ET) (μs)

Measure	DL	DR	DP
Worst-case ET	2039.6	1038.8	168875.0
Minimum ET	21.8	71.1	97596.0
Average ET	155.0	152.6	109268.7
Standard deviation	55.2	102.3	4841.7
Median	145.1	110.8	108877.0
Percentile (0.99)	286.27	394.7	122783.0

the DR and DL tasks, the DP task is the dominant in terms of WCET and therefore dwarves the other tasks.

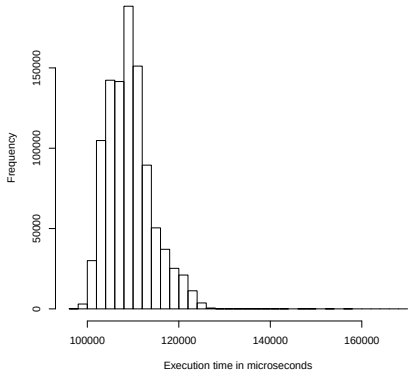
The OS used in the DP system is QNX, which allows for specification of real-time tasks, that is, critical tasks that have fixed high priorities and can pre-empt lower priority tasks. It should be noted that the tasks are independent. Since the system is periodic and all tasks have the same period, we can simply add their WCETs and compare the sum with the period/cycle length (task switching overhead is negligible in this system). In this case, the total execution time is less then $181ms$ for the 12 tasks (9 DR, 1 DL, 1 DP, and 1 IPUM tasks), corresponding to a total CPU utilization of less than 19%.

It has been shown in [25] that in Equation 2 in Section 2.2, for large values of n, the right-hand side converges to 69.3%. Since the computed DP utilization of all the tasks running in the processor add up to 18.03%, which is less than the bound of the rate-monotonic analysis, it is clear that the DP system is schedulable and all deadlines will be met.

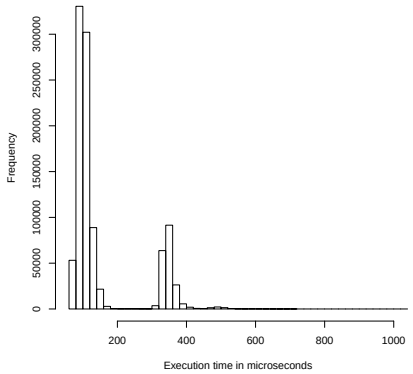
4.2 Stochastic Performance Analysis

In addition to reasoning on the worst-case execution time as presented in the previous section, we study the execution time *distribution* for the tasks on the IPU node. Studying the execution time distribution gives additional insight into the timing behavior of the system, and enables us to estimate how quickly the control loop executes in most cases (e.g. in 95% of all cases). For systems where rare misses of the deadline are acceptable, stochastic analysis can give less conservative estimates for performance and thus avoid oversizing of resources. In the DPS system, rare misses would be acceptable because the system can continue to function with the results of the previous control loop iteration.

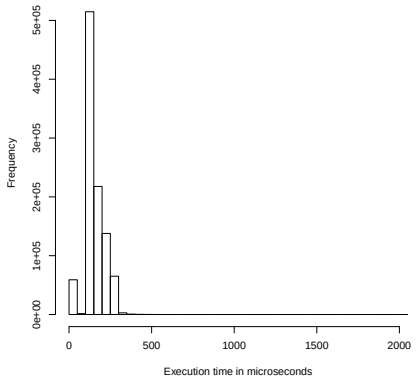
For the current system design the results from the previous section show that the control loop deadline is satisfied in the worst case. The analysis in this subsection provides additional results concerning the sensitivity of the control loop execution time to changes in the vessel configuration.



(a) Dynamic positioning (thruster allocation)



(b) Data retrieval gyro



(c) Data layer

Fig. 7 Measured execution times of tasks for stochastic performance analysis

We modeled the DataRetrieval, DataLayer, and DynamicPositioning components using measurements of the IPU tasks on a PCM model. The input data were execution time measurements for the DataRetrieval tasks of three different types of sensors (Gyro, GPS, Anemometer), for the DataLayer setDataPoint operation, and for the DynamicPositioning task, which contains the thruster allocation. We approximated the measured execution time distributions by step functions

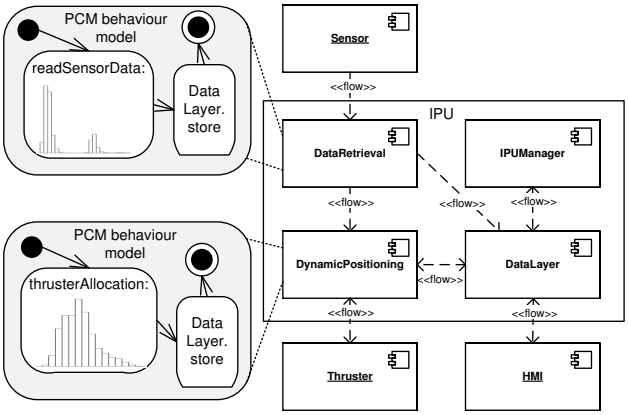


Fig. 8 Visualization of an PCM model excerpt: Measured execution times from Fig. 7 are annotated to the PCM behavior model

(some are shown in Fig. 7) and fed them into the PCM model. Figure 8 visualizes an excerpt of the resulting PCM model: each component's behavior is modeled and annotated with the measured execution time distributions.

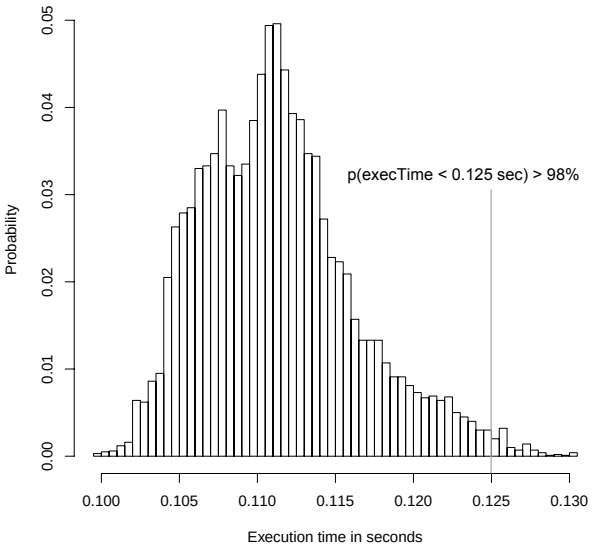


Fig. 9 Execution time distribution of the control loop with 3 sensors and 8 thrusters

Simulation results with the predicted execution time distribution are shown in Figure 9. The execution time varies between 110 ms and 130 ms. The distribution has two modes and positive skew (i.e. a longer tail on the right side). The quantiles of the distribution tell us how likely it is for the execution time to be below a given threshold. For example, the execution time is lower than 125ms in more than 98% of all cases (marked in the figure).

4.3 Sensitivity Analysis

In this section, we analyze the impact that added components have on the tasks execution times using the parametrized PCM performance models. The two studied extension scenarios are (1) new types of sensors are added for further calculations, which leads to an increase of input data and messages, and (2) the number of thrusters of the vessel varies. For both extension scenarios, we perform a sensitivity analysis by first re-evaluating the real-time performance and then determining the execution time distributions by running the PCM simulation.

If new types of sensors are added to the system (first extension scenario), more DataRetrieval tasks (one per added sensor) have to be executed on the IPU node. As every sensor is triple redundant, adding one more functional sensor would require the addition of three new physical sensors. Domain experts estimate that up to six different functional sensors could be required on a vessel, leading to up to 18 physical sensors. We assume that new types of sensors will have similar computational demands as the existing ones. In the PCM model, we vary the number of functional sensors from the currently existing three sensors to a maximum of nine sensors and study system performance. The number of thrusters is set to eight.

From a hard real-time point of view, where just WCETs are taken into account, a configuration with 8 thrusters and 9 different kinds of sensors (and thus 27 DataRetrieval tasks) would imply in a maximum processor utilization of 19.9%, and this would satisfy the condition indicated by Equation 2.

The results of the stochastic analysis are shown in Figure 10: the number of sensors has small impact on the overall execution time of the three IPU tasks. Again we observe that, even if 9 functional sensors are used, the execution time stays well below the one second deadline. Extrapolating our predictions linearly, we estimate that the critical amount of deployed sensors for the required 1 second deadline lies higher than 1000 functional sensors.

For the case that the numbers of thrusters varies (second extension scenario), the time required for the call to the thruster allocation algorithm changes. Domain experts estimate that vessels have typically from 4 to 8 thrusters.

We measured the execution time of the calls to the prototypical thruster allocation algorithm for 4, 6, and

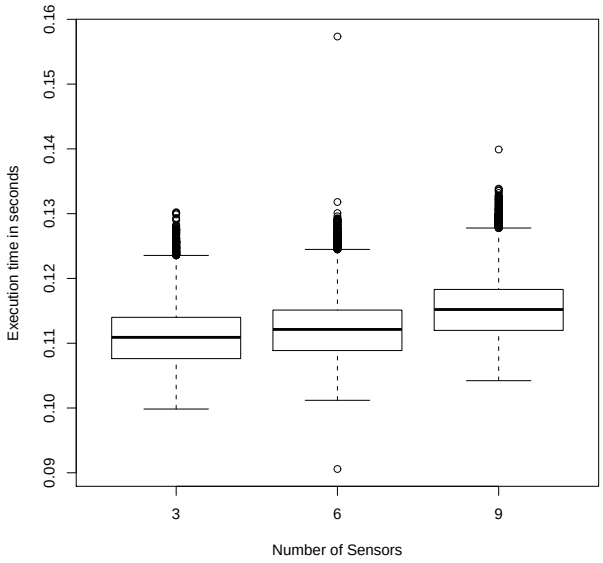


Fig. 10 The execution time distribution of the IPU tasks is shown for three different amount of functional sensors. Increasing the number of functional sensors to 6 (middle box) or 9 (right box) leads to an increased execution time, which is well below the deadline of 1 sec.

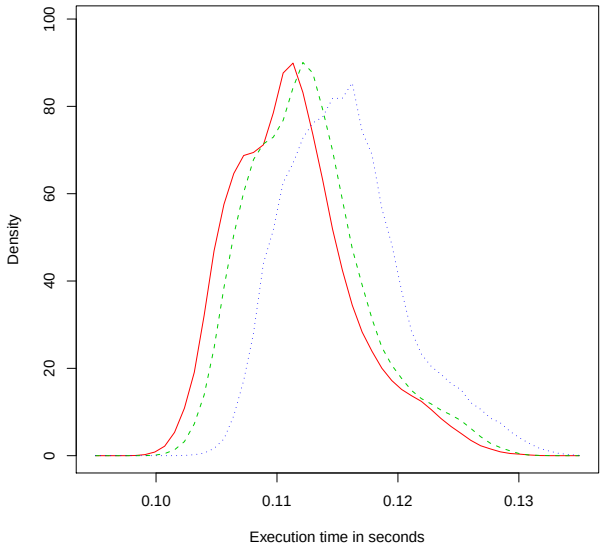


Fig. 11 Approximated density functions of sensor number change for 3 sensors (red solid line), 6 sensors (green dashed line), and 9 sensors (blue dotted line).

8 thrusters and fed the different measured execution time distributions into the PCM model. The execution time distribution as a function of the number of thrusters was estimated from the simulation model. The number of sensors is set to six.

The results of the stochastic analysis are shown in Figure 12: the number of thrusters has a higher impact on the execution time of the IPU node tasks. Thus, vessels with less thrusters than the initial configuration with eight thrusters require significantly less execution time; increasing the number of thrusters to more than eight would significantly increase execution time.

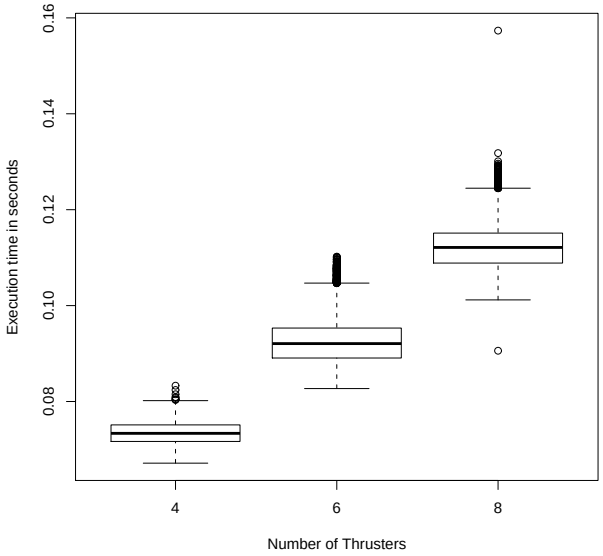


Fig. 12 Impact of thruster number change in a boxplot: The execution time distribution of the IPU tasks is shown for three different amount of thrusters. The current system has eight thrusters (rightmost box). Decreasing the number of thrusters to 6 (middle box) or 4 (left box), as expected for some vessel types, lead to an decreased execution time.

4.4 Thruster Allocation Algorithm Analysis

Sections 4.1, 4.2 and 4.3 analyze the worst-case execution time and the execution time probability distribution for the control loop of the Dynamic Positioning System (DPS). The control loop analysis is based on measurements taken from experiments conducted with a prototype system. One of the conclusions of that analysis was that the control loop execution time is impacted by the number of vessel thrusters.

In contrast, in this sub-section we simulate several realistic vessel scenarios, and analyze the number of steps required for the vessel thruster to converge to the desired position. The analysis contained in this subsection takes into account the interactions between the force that needs to be applied to turn the vessel and the number of thrusters the vessel can use to produce the required force. Therefore, the goals of this sub-section are to study the thruster algorithm convergence characteristics under different scenarios and to analyze the distribution of the force among the thrusters when the number of working thrusters varies with time. The analysis is based on a simulation model which uses the algorithm implementation.

The main goal of the algorithm is to compute the force and the angle that each thruster must have to

meet the resultant force demanded. The desired resultant force is represented by three components: (i) Surge: the vessel movement in forward or backward directions, (ii) Sway: the vessel movement in left or right directions, and (iii) Yaw: the vessel rotational movement in clockwise or counterclockwise directions on the plane formed by surge and sway. These parameters are the inputs of the algorithm.

We used the Tangram-II modeling environment [13, 16] in our evaluation. The tool provides the ability to construct from simple to complex models, and solve these by several analytical or simulation methods and to support experimentation via active measurements in computer networks. A large set of methods to calculate the measures of interest is also available. In addition, there are features that help the user to visualize the evolution of the model variables with time, useful both for developing an analytical or simulation model. A rich set of analytical solution techniques, both for steady state and transient analysis, are available, as well as, event driven and fluid simulators.

The simulation engine of Tangram-II has a feature called the modeling tool kit (MTK), which is a framework where users can develop customized algorithms as self contained plugins and use those in the simulation engine. Just like a class in the object oriented paradigm, each plugin is composed of attributes and methods, which all models created from that plugin share, and which can be accessed or executed by the user. Users can create and delete the MTK plugins (called MTK objects), set and get their attribute values, and execute their methods. In this way, users can develop complex algorithms in C++ and execute them. The simulation engine sees any MTK plugin as a black box, which corresponds to a new Tangram variable type (the MTK object).

We built a MTK plugin for the thruster allocation algorithm. The MTK plugin contains the C++ code of the algorithm implemented in the prototype system.

Table 2 shows the scenarios evaluated. The scenarios were designed to test the behavior of the algorithm in critical situations. The first six scenarios represent a sequence of vessel movements. The objective of these scenarios is to analyze the convergence time of the algorithm. In scenarios one to four, after each one of the movements, there is always a stop command. The goal is to study the algorithm behavior if the vessel stops after each movement. Scenario 6 evaluates the case when the vessel does not stop between two consecutive movements. In the seventh scenario we evaluate the behavior of the algorithm when the vessel rotates and some thrusters are turned off. Our goal is to analyze the case where some thrusters stop working.

The second column of Table 2 shows the vessel movement and the force in that direction. These are the input parameters of the algorithm. In the first scenario, for example, the resultant force demanded is represented only by the sway parameter, and in the fourth scenario, the demand is represented by the sway and surge parameters.

For each scenario and required movement, the algorithm is executed a certain number of times (It is an iterative algorithm.). The stop condition is the difference between the resultant force demanded and the resultant of the allocated force computed by the algorithm. When this difference is less or equal to 10^{-1}, the algorithm is stopped. The algorithm output is the force and the angle to be applied by each one of the thrusters and the sum of these vectors, i.e. the resultant vector.

For example, for the first movement of scenario 1 (right(40)), the input parameters were surge=0, sway=40, and yaw=0, the required number of iterations was equal to 3, and the output was surge=0.01, sway=40, and yaw=0.

Figure 13 shows the thrusters position (Cartesian coordinates) in the vessel and the thruster ID. Each thruster can produce a force which varies from 1 to 10, in unitary steps, and rotate 360, in 30 steps.

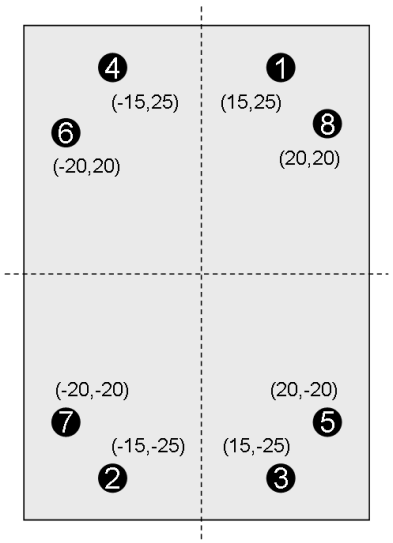


Fig. 13 Vessel's thrusters position.

We simulate each scenario 100 times and compute a 95% confidence interval. We choose not to show the confidence intervals in the figures to make the plots more readable.

Figure 14 displays the mean number of algorithm iterations for the scenarios one to six varying the number of working thrusters from 4 to 8. We consider that when the number of working thrusters is equal to n, the

Table 2 Scenarios

Scenario	Description	Required Movement(Force)
1	Left to right	right(40), left(40), right(40), left(40), right(40)
2	Forward and backward	forward(40), backwards(40), forward(40), backwards(40), forward(40)
3	Rotation	counterclockwise(10), clockwise(10), counterclockwise(10), clockwise(10)
4	Left-backwards and right-forward	forward (26.7) and right(26.7), backwards(26.7) and left(26.7), forward (26.7) and right(26.7), backwards(26.7) and left(26.7), forward (26.7) and right(26.7)
5	Keep position	stop(0)
6	All movements	left(40), right(40), forward(40),backwards(40), left(40), right(40), forward(40), backwards(40)
7	Rotation turning in and off thrusters	counterclockwise(5) during all the scenario; all th on, turn off th 8, turn off th 7 turn off th 6, turn off th 5, turn off th 4, turn off th 3, turn on th 3, turn on th 4, turn on th 5, turn on th 6, turn on th 7, turn on th 8

working thrusters are $1, 2, ..., n$ (see the thruster ID in Figure 13). For all the results in Figure 14, the length of the confidence interval is less than 6%.

Note that for the majority of scenarios, the number of iterations when only 4 thrusters are working is greater than when all thrusters are operational. The increase in the number of iterations goes from 10% (scenario 3) to 70% (scenario 1). Scenario 5, where the vessel needs only to maintain its position, is the only one where the number of iterations does not vary with the number of working thrusters. These results shows that there is a trade off between the number of working thrusters and the number of algorithm iterations. On the one hand, all thrusters must be operational to get a faster algorithm response, on the other hand, there will be more power consumption.

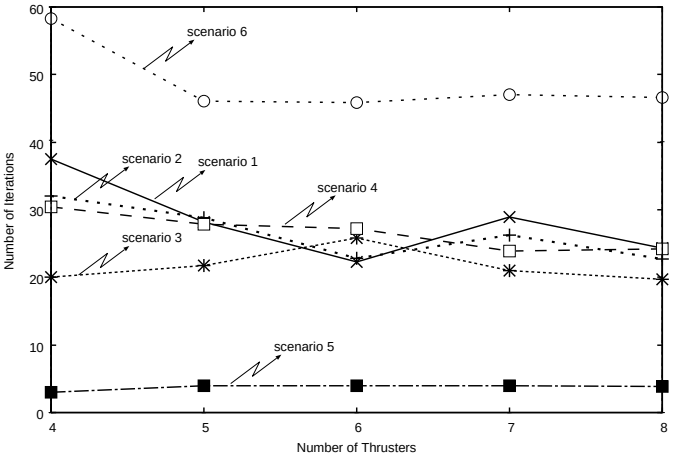


Fig. 14 Number of iterations for each scenario.

Figure 15 displays the mean force that each thruster must produce considering scenario 7, i.e., when the vessel rotates counterclockwise and each one of the thrusters is turned off and after turned on. For all the results in Figure 15, the length of the confidence interval is less than 13%.

The main goal of this analysis is to evaluate the distribution of the force among the thrusters when the number of working thrusters varies with time. Note that when only thrusters 1 and 2 are working, the force they must produce is almost the same and near the maximum value, on the other side, when all thrusters are working, each thruster have to produce a force near the minimum value. The results show that the algorithm tries to uniformly distribute the force among the working thrusters.

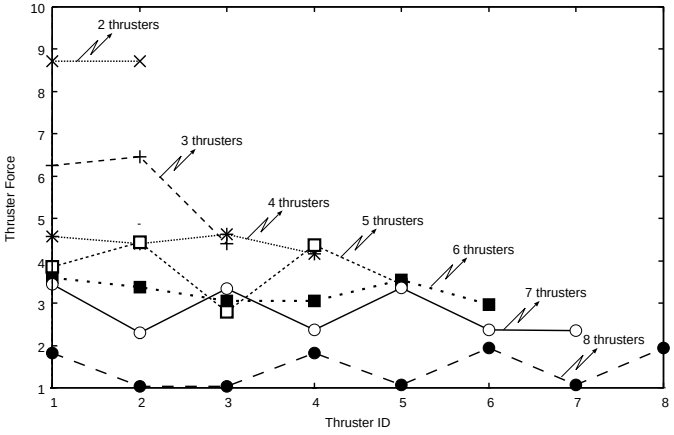


Fig. 15 Thruster force.

4.5 Lessons Learned

The three different approaches for performance analysis provided a comprehensive performance assessment of the DPS architecture.

The real-time analysis has shown that all tasks are schedulable. The sensitivity analysis of tasks schedulability was performed by increasing the number of thrusters and sensors. Even for these larger configurations we have shown that all the tasks are schedulable and all deadlines will be met, as shown in Section 4.3.

The control loop execution time is impacted when control loop components are extended with additional functionality. The stochastic performance analysis can

be used to assess the impact of component changes on the control loop execution time.

We have shown in this paper that embedded real-time systems can be modeled with the PCM component-based approach, when task precedence can be simplified into to a task sequence. In addition, stochastic performance analysis can complement worst-case predictions. However, tasks with more complex scheduling behavior could not be predicted with the PCM approach without taking into account task priorities.

We used the Tangram-II tool to simulate several realistic vessel scenarios, and analyze the number of steps required for the vessel thruster to converge to the desired position. One distinct feature of the approach is to allow the use of the real implementation code as a black box and embedded in the simulator tool. Thus, we can use the full power of the simulation engine to test the code.

5 Adaptability and Evolution Assessment

It is highly possible that the requirements of the system will change. For example, the program may need to be deployed in different types of vessels or a new type of sensor needs to be installed. We need to assess if the architecture is adaptive enough so that these new features can be accommodated quickly. There can be business metrics to assess the time needed to deliver these new features, for example, by specifying the maximum number of months that can be spent on implementing, testing, and deploying new features. In order to make such estimations, we first need to calculate which and how many components will be added, deleted, or modified, given a specified change. Neither design-level component diagrams nor the source code support such calculation directly. We thus leverage the *design structure matrix* (DSM) [4] model to achieve this purpose. We created DSMs to represent both the design architecture and the implemented structure for each release.

The DSMs shown in Figure 16, 17 and 18 are all derived from ACN models. The design ACN model is transformed from the component diagrams modeled using a commercial tool called Enterprise Architect (EA), and the source DSM is transformed from the implemented source code. These DSMs represent different ways the modular structure of the same system can be modeled. A *module* in a DSM is a group of variables that can be represented as a block along the diagonal, or clustered and represented as a compound variable in a DSM. In Figure 16 and 17, for example, elements 5 (Chemtech.DP.DataPoints) to 11 (Chemtech.DP.HMI) represent modules clustered according to the namespaces. For example, Chemtech.DP.HMI contains all the components within the Chemtech.DP.HMI namespace. In Figure 18, blocks along the diagonal are the modules automatically aggregated using our design rule hierarchy algorithm [32].

We use the design rule hierarchy in Figure 18 to assess the designed adaptability. The top level of the hierarchy (element 1-10) are the top level *design rules* that only influence other parts of the system but are not influenced by them. As we can see, the top level contains the communication infrastructure and the abstract interfaces for sensors and devices. The figure also shows that in the designed architecture, elements that have significant, cross-cutting influences are all at the first two layers, which are mainly communication infrastructure or the high-level architecture framework. About 72% (*Independence Level* [31]) of the components are in the third layer, which means that these modules can be implemented and changed independently from each other, as long as the design rules in the first two layers are stable.

As indicated in our prior work [31], the more layers are there in the system, the worse it is modularized because more parts of the system are vertically constrained. The larger the last layer, the better it is modularized because more parts of the system can evolve independently from each other. To assess how well the design can accommodate particular changes, such as adding or changing a sensor or adding an error masking mechanism to the vessel control system, we just need to see which and how many other components will be affected. From the design DSM, we can see that changes to sensors only affect at most two other components, and adding error masking mechanism only affects the main function. As another example, if a YoungSerial device is added or changed, then only the YoungDataRetrieval component will be affected. From these analyses, we conclude that the system is designed to be well modularized and adaptive to these envisioned changes.

Next we assess whether the implementation of the system is consistent with the design and maintains the same level of adaptability. Figure 17 shows DSM modeling the first release of the source code, clustered in the same way as the design DSM (Figure 16). In this DSM, the cells with dark background indicate the discrepancies between design and implementation. If a dark cell is empty, it means the dependency exists in design, but not in implementation. If a dark cell is not empty, it means that dependency in the source code does not exist in design. Figure 17 shows the discrepancies among clusters. The numbers in the cell are the total number

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1 Chemtech.DP.IPC.IPC	.																					
2 Chemtech.DP.CommunicationManager.CommunicationManager	1	.																				
3 Chemtech.DP.NetDriver.NetUDP_CPlusPlus			.																			
4 Chemtech.DP.NetDriver.NetUDP_CSharp				.																		
5 Chemtech.DP.DataPoints					.																	
6 Chemtech.DP.DataPoointClient		1		1		.																
7 Chemtech.DP.Sensors							.															
8 Chemtech.DP.Devices								.														
9 Chemtech.DP.DataPointMessages									.													
10 Chemtech.DP.Consolidators										.												
11 Chemtech.DP.HMI						1					.											
12 Chemtech.DP.ThrusterAllocation.ThrusterAllocation												.										
13 Chemtech.DP.GPSDataRetrieval.GPSDataRetrieval		1				1		2					.									
14 Chemtech.DP.GyroDataRetrieval.GyroDataRetrieval		1				1		2						.								
15 Chemtech.DP.KalmanFilter.KalmanFilter															.							
16 Chemtech.DP.DataLayer.DataLayer		1	1		1											.						
17 Chemtech.DP.IPUManager.MasterDiscovery			1			1											.					
18 Chemtech.DP.VesselControl.VesselControl																		.				
19 Chemtech.DP.YoungDataRetrieval.YoungDataRetrieval		1				1		2											.			
20 Chemtech.DP.DynamicPositioning.DynamicPositioning		1				1						1			1			1		.		
21 Chemtech.DP.FakeGps.FakeGps																					.	
22 Chemtech.DP.IPUManager.IPUManager			1													1						.

Fig. 16 Design level DSM

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1 Chemtech_DP_IPC_IPC_CPlusPlus	.																			
2 Chemtech_DP_CommunicationManager_CommunicationManager_CPlusPlus	1	.																		
3 Chemtech_DP_NetDriver_NetUDP_CPlusPlus			.																	
4 Chemtech_DP_NetDriverClient_NetUDP_CSharp				.																
5 Chemtech.DP.DataPoints					.															
6 Chemtech.DP.Sensors						.														
7 Chemtech.DP.Devices						8	.													
8 Chemtech.DP.DataPointMessages					4			.												
9 Chemtech.DP.Consolidators						6			.											
10 Chemtech.DP.DataPointClient		1		1	1		2			.										
11 Chemtech.DP.HMI										10	.									
12 Chemtech_DP_DataLayer_DataLayer_CPlusPlus		1	1									.								
13 Chemtech_DP_IPUManager_MasterDiscovery_CPlusPlus		1	1							1			.							
14 Chemtech_DP_GPSDataRetrieval_GPSDataRetrieval_CPlusPlus		1			1		2			1				.						
15 Chemtech_DP_GyroDataRetrieval_GyroDataRetrieval_CPlusPlus		1			1		2			1					.					
16 Chemtech_DP_KalmanFilter_KalmanFilter_CPlusPlus																.				
17 Chemtech_DP_YoungDataRetrieval_YoungDataRetrieval_CPlusPlus		1			1		2			1							.			
18 Chemtech_DP_VessalControl_VessalControl_CPlusPlus																		.		
19 Chemtech_DP_ThrusterAllocation_ThrusterAllocation_CPlusPlus																			.	
20 Chemtech_DP_DynamicPositioning_DynamicPositioning_CPlusPlus		1				3		2	1								1		1	1 .

Fig. 17 Source level DSM

of dependencies between modules clustered according to source code namespace.

The analysis shows that the main discrepancies are that in the implementation the data points and sensors are accessed by more components than designed. These discrepancies are caused by the following reasons: more dependencies are found to be necessary during implementation than recorded in the component diagram, the system have evolved in code but the design is not updated, or a part of the design has not been implemented yet. However, the majority of the source code realized the design faithfully. When comparing the source code DSM to the design DSM, we found that the source code also has a layered structure, and each type of sensor only had at most one more dependency than designed. Basically, the first version of implementation indicates similar level of modularity and adaptability between design and implementation.

It is common practice for the system implementation to keep evolving. We studied five consecutive releases of the system to assess (1) if the implementation has deviated from the design; (2) how the system adaptability changes over time. Our approach was to compare the source code DSMs from release 2 to release 5, against the DSM of release 1. We checked how many and at which level components were added, deleted or changed. We also calculate how the independence level varies over these releases. We summarize the results as follows:

Release 2 has the same three layers as release 1. Two top-level design rules were added to the top layer, seven interfaces were added to the second layer, and fifteen components were added to the bottom layer. Therefore, the architecture of the system has changed from release 1. Comparing with release 2, release 3 just has minor changes: one component was moved from layer 3 to layer 2 and four new dependencies were added. From release 3 to release 5, the top 2 layers were not changed. Fifteen new components were added in release 4, and four more components were added in release 5. Besides the dependencies added with the new components, eight new dependencies were added and

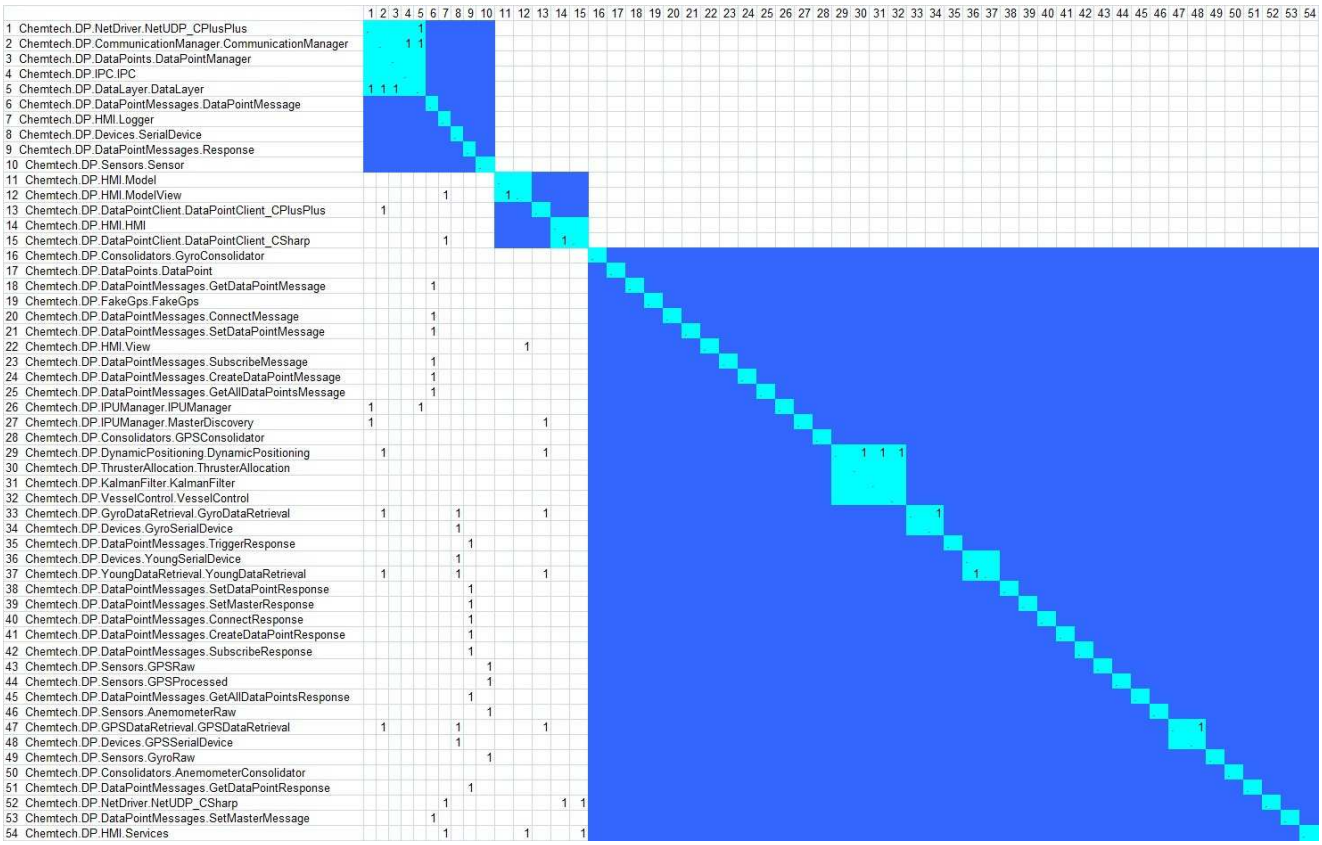


Fig. 18 Design DSM expanded

two existing dependencies were deleted. We also clustered these five DSMs using design rule hierarchy and calculated their Independence Levels, which are 76.5%, 72.8%, 71.7%, 75.7%, 75.7% from release 1 to release 5.

As a result of this analysis we make the following observations. First, although the architectural-level design rules of the system have changed, especially in release 2, the architecture component diagram was not updated. Ideally, when valid changes are made to the source code the component diagram should be updated as well. We recommend that valid changes should be initiated at the component diagram and be followed by an evaluation step, before being propagated to implementation. Second, several components were added to the source code but not to the design. As a result, it is hard to tell if these changes are valid. Finally, although the architecture has been changed, the adaptability of the system was not degraded significantly. Except for release 2, components were added to the bottom layer of the DSM without significant impact on the existing components or the design rules.

6 Reliability/Availability

6.1 Hardware

In this section we analyze the reliability of the overall DP system hardware architecture ($R_h(t)$). Our goal is to obtain the repair rates that satisfy the condition: *system reliability is greater than 0.9999 for a mission time equal to 3 months*. In other words, we compute the probability of the DP system being operational at t = 3 months for some values of repair rates. We select the repair rates where $R_h(3) \geq 0.9999$.

We created a Tangram-II hardware model shown in Figure 19 for the hardware reliability analysis. The model has five different components: gyro sensor, GPS, anemometer, IPU and local network. All components, except the local network (which is dual redundant), are triple redundant.

Figure 20 shows the reliability of the system in the interval from 1 hour to 3 months. We note that for the repair rates equal to 1 day and 15 days, the probability of the DP system being operational at t = 3 months is 0.999999 and 0.9999, respectively. For the repair rates equal to one and three months, $R_h(3) \leq 0.9999$. We

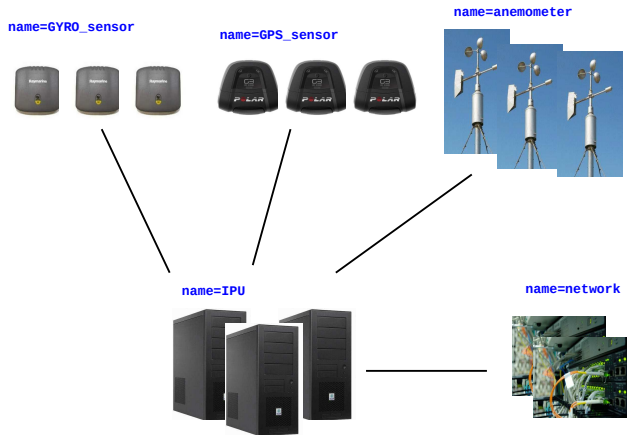


Fig. 19 System reliability model.

conclude that the system reliability goal is met for a repair rate less or equal to 15 days.

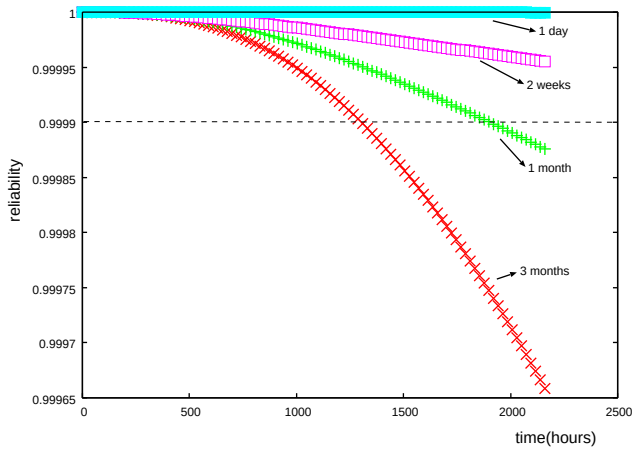


Fig. 20 System reliability.

6.2 Software

The software reliability was tested using a black-box approach. A simulator was developed by the Siemens-Chemtech team to test the DPS without the need of a real vessel. The simulator mimics the vessel and the environment where it is immersed. Figure 21 shows how the simulator replaces the sensors and the thrusters, providing the information flow that should come from the sensors and the thrusters and consuming the information flow that should go to the thrusters.

There are three main environmental forces that act on the vessel: wind, current and tides. The simulator models each one of the environmental forces numerically. The models are configured a priori according to the desired scenario parameters. In addition to the environmental forces, the simulator considers the allocation of each thruster required by the DPS. The positioning and environmental data are sent to the DPS using the default protocols of each kind of sensor. Additionally, noise is applied to each sensor instance.

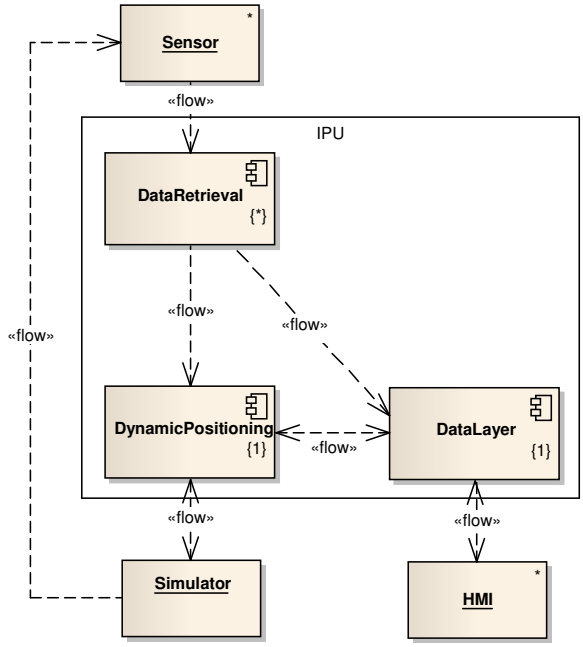


Fig. 21 IPU information flow with simulator

The simulator provides functionality to configure alarms when the vessel loses its position. There are two levels of alarms to deviations in the surge, sway and heading. Any time the vessel surpass one of these thresholds, one alarm is opened, and when the vessel returns to the designated position, the alarm is closed and logged. The second level alarms are also referred to as critical alarms.

The first simulation run imposed very stringent position and heading constraints, because the DPS first priority is to keep the vessel position. The surge and sway alarms were set to 1 meter and the associated critical alarms were set to 2 meters. The heading alarm was set to 1 degree and its critical alarm was set to 2 degrees. The simulation was executed for 160000 seconds (1 day, 20 hours, 26 minutes and 40 seconds) and the cumulative alarm count per alarm type is shown in Figure 22. No heading alarms were generated in this run.

The first simulation run accounted 4.1625 first level alarms per hour and 0.315 second level alarms per hour. For the first level alarms, 76% were surge alarms and 24% were sway alarms. For the second level alarms, 71% were surge alarms and 29% were sway alarms. In this simulation, as the vessel has its bow facing towards the direction of the greatest environmental force, it is

expected that there is more displacement in the surge axis, even if the demanded thruster allocation is met. In this case, the sway displacement is a consequence of the vessel's surge position adjustment. The algorithm prioritizes the maintenance of the heading, hence no heading alarms occurred in this run.

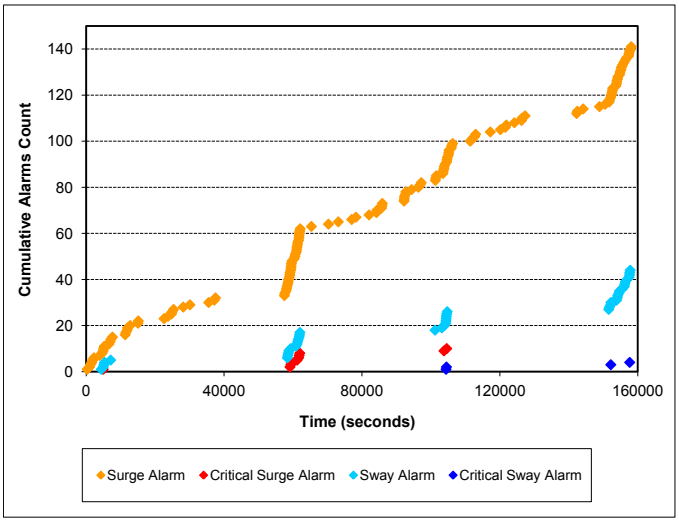


Fig. 22 Black-box first simulation run results

The second simulation run imposed more usual position and heading constraints in the same scenario of first run. The surge and sway alarms were set to 2 meters and the associated critical alarms were set to 4 meters. The heading alarm was set to 1 degree and its critical alarm was set to 2 degrees. The simulation was executed for 110000 seconds (1 day, 6 hours, 33 minutes and 20 seconds) and the cumulative alarm count per alarm type is shown in Figure 23.

The second simulation run accounted 0.2618 first level alarms per hour and 0.0655 second level alarms per hour. For the first level alarms, 62.5% were surge alarms, 25% were sway alarms and 12.5% were heading alarms. These results are consistent with the first run results, as the first level alarms for positioning in the second run are equal to the second level alarms for positioning in the first run. A lower occurence of second level alarms was also expected due to the more relaxed constraints. The distribution of the alarms in terms of position and heading are also consistent with the first run.

There are 3 well defined alarms groupings in the second run: one surge alarm in the time 2645, one surge alarm in the time 18478 and a group of alarms ranging from time 90540 to time 92028. This third group of alarms should be considered as an event where the position and heading were lost for a period of 1488 seconds (24 minutes and 48 seconds). In other words, there were 3 failure events in the simulation, accounting 0.0982 failures per hour or 2.3568 failures per day.

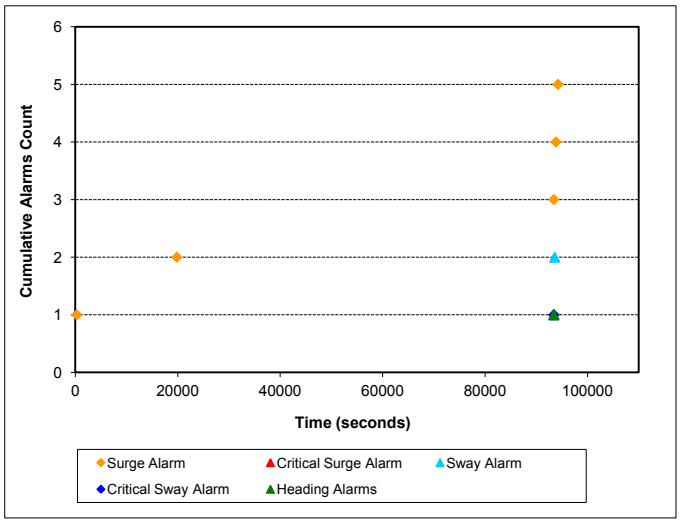


Fig. 23 Black-box second simulation run results

The DPS is still a prototype system and further adjustments to the modules' tuning should improve their response to the environmental disturbances, increasing the system reliability. For future work, improved versions of the system should be tested and it is also desirable to test the DPS in other simulators to compare the reliability results.

7 Conclusions

We have presented several performance, reliability and adaptability models that were used to comprehensively assess the Dynamic Positioning System architecture. The three performance models presented were instrumented using data collected from the Siemens-Chemtech software prototype. These modeling activities were conducted after an extensive system architecture review uncovered several architecture risks. These risks were reported in a companion paper [14]. The results obtained from the experiments using the non-functional requirement models presented in this paper were of great value to the project in several ways. As a result of the extensive performance modeling experiments reported in this paper, the team has now an increased understanding of the system's ability to meet its real-time deadlines, of the impact of different system configurations on the control loop execution time distribution, and the system configurations impact on the convergence characteristics of the thruster allocation algorithm. The real-time analysis has shown that all

tasks are schedulable and that the system will meet its deadlines. The Tangram-II implementation based simulation approach was able to execute several tests using the thruster allocation module and confirmed that the thruster algorithm is stable and generates a well-balanced mean force allocation. The Tangram-II based framework could be used to execute several additional test scenarios to further test the Dynamic Positioning System. A black-box reliability testing approach of the implemented system was executed using a simulation environment. Failure rates were derived for two different set-points for two long running simulation experiments. Reliability data was collected and recommendations for reliability improvement were provided to the development team. As a result we were able to estimate the Dynamic Positioning System software reliability using actual system testing results.

In addition, as a result of the system architecture adaptability assessment, the project has received feedback on the importance of maintaining the system architecture documentation up to date. The adaptability assessment model has shown that the system is very modularized, has a layered structure, and that the system implementation complies with the guidelines provided by the system architecture document. This assessment certifies that the system architecture is adaptable and extensible.

8 Acknowledgments

We thank FINEP and CNPq for partial financial support of the project.

References

1. W. Abdelmoez, D. E. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, B. Yu, and A. Mili. Error propagation in software architectures. In *IEEE METRICS*, pages 384–393. IEEE Computer Society, 2004.
2. N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Preemptive Scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.
3. T. P. Baker. A stack-based resource allocation policy for real-time processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.
4. C. Y. Baldwin and K. B. Clark. *Design Rules, Volume 1: The Power of Modularity*. MIT Press, Cambridge, MA, USA, 2000.
5. S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.
6. S. Bernardi, J. Merseguer, and D. Petriu. A dependability profile within marte. *Software and Systems Modeling*, pages 1–24, 2009. 10.1007/s10270-009-0128-1.
7. A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Mul tiprocessor Systems. *IEEE Trans. Comput.*, 44(12):1429–1442, 1995.
8. A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.
9. Y. Cai. *Modularity in Design: Formal Modeling and Automated Analysis*. PhD thesis, University of Virginia, August 2006.
10. Y. Cai and K. Sullivan. A formal model for automated software modularity and evolvability analysis. *ACM Transactions on Software Engineering and Methodology*.
11. V. Cortellessa and V. Grassi. A modeling approach to analyze the impact of error propagation on reliability of component-based systems. In H. W. Schmidt, I. Crnkovic, G. T. Heineman, and J. A. Stafford, editors, *Proceedings of the 10th International Conference on Component-based Software Engineering*, volume 4608 of *Lecture Notes in Computer Science*, pages 140–156. Springer, 2007.
12. V. Cortellessa, H. Singh, and B. Cukic. Early reliability assessment of UML based software models. In *Workshop on Software and Performance*, pages 302–309, 2002.
13. E. de Souza e Silva, D. R. Figueiredo, and R. M. Leão. The TANGRAMII integrated modeling environment for computer systems and networks. *SIGMETRICS Perform. Eval. Rev.*, 36(4):64–69, 2009.
14. F. Duarte, C. Pires, C. A. de Souza, J. P. Ros, R. M. M. Leão, E. de Souza e Silva, J. C. B. Leite, V. Cortellessa, D. Mosse, and Y. Cai. Experience with a new architecture review process using a globally distributed architecture review team. In *The 5th IEEE International Conference on Global Software Engineering (ICGSE 2010)*, pages 109–118, Los Alamitos, CA, USA, 2010. IEEE Computer Society.
15. J. B. Dugan and K. S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Trans. Computers*, 38(6):775–787, 1989.
16. Federal University of Rio de Janeiro. Tangram-II website. http://www.land.ufrj.br/tools/tangram2/tangram2.html, 2010.
17. Gilson A. Pinto et al. Advanced control and optimization techniques applied to dynamic positioning systems. In *Rio Oil & Gas Expo and Conference*, Sept. 2010. in press.
18. A. L. Goel and K. Okumoto. Time-dependent error-detection rate model for software reliability and other performance measures. *Reliability, IEEE Transactions on*, R-28(3):206 –211, aug. 1979.
19. S. Gokhale and K. Trivedi. Analytical models for architecture-based software reliability prediction: A unification framework. *Reliability, IEEE Transactions on*, 55(4):578 –590, dec. 2006.
20. D. D. Gouvêa, C. Muniz, G. Pinto, A. Avritzer, R. M. M. Leão, E. de Souza e Silva, M. C. Diniz, L. Berardinelli, J. C. B. Leite, D. Mossé, Y. Cai, M. Dalton, L. Kapova, and A. Koziolek. Experience building non-functional requirement models of a complex industrial architecture. In *Proceeding of the second joint WOSP/SIPEW international conference on Performance engineering (ICPE'2011)*, 2011.
21. L. Kapova and R. Reussner. Application of advanced model-driven techniques in performance engineering. In A. Aldini, M. Bernardo, L. Bononi, and V. Cortellessa, editors, *Computer Performance Engineering*, volume 6342 of *Lecture Notes in Computer Science*, pages 17–36. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15784-4_2.

22. R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *Software, IEEE*, 13(6):47–55, nov 1996.
23. H. Koziolek. *Parameter Dependencies for Reusable Performance Specifications of Software Components*, volume 2 of *The Karlsruhe Series on Software Design and Quality*. Universitätsverlag Karlsruhe, 2008.
24. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling: Exact characterization and average case behavior. *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.
25. C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):47–61, 1973.
26. J. Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14 –32, mar 1993.
27. Object Management Group (OMG). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems (formal/2009-11-02). http://www.omgmarte.org/, 2009.
28. QNX Software Systems. QNX Neutrino RTOS. http://www.qnx.com/products/neutrino-rtos/ neutrino-rtos.html, 2010.
29. R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium*, pages 259–269, 1988.
30. N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proc. 20th*, pages 167–176, Oct. 2005.
31. K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *Joint 8th Working IEEE/IFIP Conference on Software Architecture and 3rd European Conference on Software Architecture (WICSA/ECSA)*, pages 269–272, 2009.
32. G. V. G. S. Sunny Wong, Yuanfang Cai and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th. IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208. IEEE Computer Society, 2009.
33. S. Wong and Y. Cai. Improving the efficiency of dependency analysis in logical decision models. In *Proc. 24th. IEEE/ACM International Conference on Automated Software Engineering*, pages 173–184. IEEE Computer Society, 2009.